GPU Server Sizing Methodology for LLM Inference Solutions: A Formalized Approach to Infrastructure Capacity Planning

Bogdan Garbar, Dmitry Velibekov, Pavel Sozonov,
Pavel Lukash, Yaroslav Kotov

OTP Bank, Moscow, Russia
Correspondence: bvgarbar@gmail.com
Inference Calculator Sources on GitHub

March 2026

Abstract

In the context of high capital intensity of AI (LLM) inference hardware, traditional heuristic infrastructure estimation methods become economically inefficient. This work proposes a formalized approach to infrastructure sizing for large language model deployment. The obtained calculation results serve as input data for forming a predictable capital expenditure (CapEx) model, enabling businesses to improve the accuracy of Total Cost of Ownership (TCO) estimation and Return on Investment (ROI) assessment.

To account for factors that are difficult to describe analytically with precision (framework overhead, pipeline specifics, request length distribution, network latency, etc.), the model incorporates calibration (correction) coefficients. Their values are determined based on pilot load testing and operational telemetry, and are regularly reviewed as statistics accumulate. This approach allows for refinement of calculations while maintaining transparency of assumptions and reproducibility of results.

The document presents a comprehensive methodology for calculating infrastructure requirements for deploying large language models (LLMs) in production environments. The methodology accounts for both memory constraints (model weights and KV-cache) and computational throughput requirements, providing a systematic approach to determining server cluster size. Modern advances in LLM optimization, including quantization and Tensor Parallelism, require a more sophisticated approach to capacity planning compared to traditional methods.

Keywords: LLM inference, GPU capacity planning, Tensor Parallelism, KV-cache, infrastructure sizing, performance modeling, SLA compliance

Author Contributions

Bogdan Garbar, Chief Product Owner AI Platform, OTP Bank, Moscow: Problem formulation, methodology development and formalization, preparation of examples and appendices, calculator development.

Dmitry Velibekov, AI Tech Lead (ML), OTP Bank, Moscow: Methodology development and formalization, preparation of examples and appendices, calculator development.

Pavel Sozonov, AI Tech Lead (LLM), OTP Bank, Moscow: Methodology development and formalization, preparation of examples and appendices, calculator development.

Pavel Lukash, AI Software Developer (LLM), OTP Bank, Moscow: Methodology development and formalization, preparation of examples and appendices, calculator development.

Yaroslav Kotov, AI Software Developer (ML), OTP Bank, Moscow: Methodology development and formalization, preparation of examples and appendices, calculator development.

Contents

1 Brief Methodology Overview **4**

2 Load Definition in Users and Tokens **5**

2.1 Number of Simultaneously Active Users 5

2.2 Upper Estimate of Average Context Length per Session 6

3 Determining Required GPU Memory for Model and KV-Cache **7**

3.1 Memory Required for Model Weights 7

3.2 KV-Cache per Session 7

4 GPU Memory Optimization with Tensor Parallelism (TP) **9**

4.1 Minimum Required Number of GPUs 9

4.2 Number of Model Instances per Server 10

4.3 Free GPU Memory for KV-Cache per Model Instance 11

4.4 Maximum Theoretical Number of Parallel Sessions 11

5 Determining Number of Servers by GPU Memory **12**

5.1 Throughput of One Server by Memory 12

5.2 Number of Servers by Memory 12

6 Determining Number of Servers by GPU Computational Throughput **13**

6.1 Throughput of One Model Instance 13

6.2 Average Number of Simultaneous Requests per Second per Model Instance 16

6.3 Final Throughput of One Server 16

6.4 Number of Servers by Throughput 17

7 Verification of Selected Server Configurations by TTFT and e2eLatency **18**

7.1 Time To First Token (TTFT) Calculation 18

7.2 Latency Indicator Determination 18

7.3 Comparison of TTFT and e2eLatency with Target SLA Values 19

8 Determining Final Number of Servers **19**

A Derivation of Throughput Gain Calculation from Tensor Parallelism **22**

B Recommendations for Aligning Configuration with Target TTFT and e2eLatency Values **24**

B.1 Increase TP Multiplier (Z) 24

B.2 Apply More Aggressive Quantization 24

B.3 Reduce Context Length (SL) . . . 25
B.4 Reduce Generation Volume (T_{out}) . . . 25
B.5 Change Model . . . 25
B.6 Use More Performant Hardware . . . 26
B.7 Reconsider Target SLA Values . . . 26
B.8 Measure Selection Matrix . . . 27

C Recommendations for Token Load Estimation Strategy Depending on AI Solution Architecture 28

C.1 Connection with Main Methodology Formulas . . . 28
C.2 What Changes in Agent Architectures . . . 28
C.3 Generalized Formula for Agent Architectures . . . 29
C.4 Architectural Pattern Catalog . . . 29
C.4.1 Single-turn Chatbot (Basic Case) . . . 29
C.4.2 RAG (Retrieval-Augmented Generation) . . . 29
C.4.3 Chain-of-Thought (CoT) and Reasoning Models . . . 30
C.4.4 Self-Refine / Iterative Improvement . . . 30
C.4.5 ReAct (Agent with Tools) . . . 30
C.4.6 Multi-Agent Systems . . . 30
C.4.7 Function Calling / Tool Use . . . 30
C.5 Context Compounding Effect . . . 31
C.6 Summary Pattern Table . . . 31
C.7 Calculation Examples . . . 32
C.7.1 Example 1: Simple RAG Chat . . . 32
C.7.2 Example 2: ReAct Agent with Tools . . . 32
C.7.3 Example 3: Multi-Agent Team . . . 32
C.7.4 Example 2B: ReAct Agent with Compounding . . . 33
C.8 Impact on Throughput Calculation . . . 33
C.9 Strategies for Reducing Token Load . . . 34
C.10 Evaluation Algorithm and Measurement Recommendations . . . 35

1 Brief Methodology Overview

In the context of high capital intensity of AI (LLM) inference hardware, traditional heuristic infrastructure estimation methods become economically inefficient. This work proposes a formalized approach to infrastructure sizing. The obtained calculation results serve as input data for forming a predictable capital expenditure (CapEx) model, enabling businesses to improve the accuracy of Total Cost of Ownership (TCO) estimation and Return on Investment (ROI) assessment.

To account for factors that are difficult to describe analytically with precision (framework overhead, pipeline specifics, request length distribution, network latency, etc.), the model incorporates calibration (correction) coefficients. Their values are determined based on pilot load testing and operational telemetry, and are regularly reviewed as statistics accumulate. This allows for refinement of calculations while maintaining transparency of assumptions and reproducibility of results.

This document presents a comprehensive methodology for calculating infrastructure requirements for deploying large language models (LLMs) in production environments. The methodology accounts for both memory constraints (model weights and KV-cache) and computational throughput requirements, providing a systematic approach to determining server cluster size. Modern advances in LLM optimization, including quantization and Tensor Parallelism, require a more sophisticated approach to capacity planning compared to traditional methods.

Online usage: You can use our methodology by entering online Inference Calculator App on https://calc.aicolab.space/

Recommendation: Apply the methodology separately for each specific segment. Within this methodology, a segment is defined as a group of users united by common processes and expected outcomes (e.g., specific internal divisions such as call centers, or separate external groups such as premium clients). Accordingly, calculations should be performed in isolation for each such subgroup to ensure accuracy of the final result.

2 Load Definition in Users and Tokens

2.1 Number of Simultaneously Active Users

User count calculation is performed separately for each segment. In particular, if there is a division between external and internal users (examples: full-time employees and contractors, in-house specialists and vendor/customer representatives), calculation is performed for the internal segment first, then for the external segment.

$$S_{sim} = N_{users} \times K_{pen} \times K_{sim} \times J \tag{1}$$

Table 1: Variables for simultaneous active users calculation

Variable	Definition	Unit	Default	Notes
S_{sim}	Peak number of simultaneous employee sessions	sessions	–	Calculation is assumed to be performed for each separate user segment if needed. Example: internal users (e.g., bank employees) and external users (e.g., vendor employees)
N_{users}	Number of users from selected segment	users	–	–
K_{pen}	Penetration coefficient among users (share using GenAI platform)	float	–	Determined based on empirical data separately for each user group
K_{sim}	Simultaneity coefficient for internal users	float	–	Determined based on empirical data separately for each user group
J	Number of simultaneous sessions per user	sessions/user	1	–

2.2 Upper Estimate of Average Context Length per Session

$$TS_{N_{prp}}^{S=1} = SP + N_{prp} \times (Prp + MRT + A) \tag{2}$$

Table 2: Variables for context length estimation

Variable	Definition	Unit	Default	Notes
$TS_{N_{prp}}^{S=1}$	Estimated context length in 1 session accounting for number of requests and responses	tokens	–	This variable can be regarded as an upper estimate of average dialogue history. The upper bound is determined by parameter N_{prp}
SP	Number of tokens in system prompt	tokens	1000	Empirical research required regarding target request for capacity calculation
N_{prp}	Number of request-response pairs	int	5	Assumed for dialogue of 5 messages as expert estimate (number 5 can be changed if needed)
Prp	Number of tokens in user request	tokens	200	Average 500 characters
A	Number of tokens in model response	tokens	400	Average 900 characters
MRT	Token budget for reasoning if model uses it	tokens	4096	If model does not use internal reasoning or it is disabled, set $MRT = 0$. For estimation, better to specify expected average value (e.g., 20-50% of maximum budget) rather than maximum

Note: See Appendix C for details.

3 Determining Required GPU Memory for Model and KV-Cache

3.1 Memory Required for Model Weights

$$M_{model} = \frac{P \times 10^9 \times B_{quant}}{1024^3} \times EMP_{model} + SM \tag{3}$$

Table 3: Variables for model weights memory calculation

Variable	Definition	Unit	Default	Notes
M_{model}	Memory required for model weights	GiB	–	–
P	Number of model parameters	int	–	Example: 7B, 13B, 70B
B_{quant}	Bytes per parameter	bytes	–	FP8 $\rightarrow$ 1; FP16 $\rightarrow$ 2; FP32 $\rightarrow$ 4
SM	Safe margin, safe memory offset	GB	5	Accounts for framework service buffers, memory alignment/fragmentation, CUDA graphs, workspaces, and other overhead of placement on any inference service (vLLM, Ollama, TensorRT, etc.)
EMP_{model}	Correction coefficient for practical occupied memory	float	1	Allows accounting for differences between "ideal" parameter-based calculation and actual footprint in specific stack (vLLM/TensorRT-LLM, weight format, loader, allocator). If no data available, use 1.0–1.15 (conservative)

3.2 KV-Cache per Session

$$SL = \min\left(TS_{N_{prp}}^{S=1}, TS_{max}\right) \tag{4}$$

$$M_{KV}^{S=1} = \frac{2 \times L \times H \times SL \times B_{state} \times EMP_{kv}}{1024^3} \times \frac{N_{kv}}{N_{attention}} \tag{5}$$

Table 4: Variables for KV-cache memory calculation

Variable	Definition	Unit	Default	Notes
$M_{KV}^{S=1}$	Memory for KV-cache per 1 session	GiB	–	Shows how much memory must be reserved before session completes or context is moved to storage other than GPU memory (e.g., vector DB)
L	Number of model layers	layers	–	Depends on architecture. Usually specified in model specification
H	Hidden state dimension	int	–	Model hidden state size. Depends on architecture. Usually specified in model specification
$TS_{N_{prp}}^{S=1}$	Estimated context length in 1 session accounting for requests/responses	tokens	–	See Section 2.2
TS_{max}	Maximum context window	tokens	–	Taken from model specification
B_{state}	Bytes per state value	bytes	2	FP8 $\rightarrow$ 1; FP16 $\rightarrow$ 2; FP32 $\rightarrow$ 4
EMP_{kv}	Correction coefficient for practical KV-cache size	float	1	Accounts for KV-cache implementation (paged attention, blocks/pages, alignment), fragmentation, and service structures. If no data available, use 1.0–1.2 (conservative)
N_{kv}	Number of KV-cache "heads"	int	32	Number of vectors sent to transformer "head", usually equals number of transformer heads
$N_{attention}$	Number of transformer "heads"	int	32	Number of independent, parallel scalable scalar products in transformer attention mechanism
SL	Context window sequence length	tokens	–	With paged attention, KV-cache memory grows based on actually used tokens; without it, max_model_len (maximum context window) is often reserved

4 GPU Memory Optimization with Tensor Parallelism (TP)

4.1 Minimum Required Number of GPUs

$$GPU_{count_{min}}^{model=1} = \left\lceil \frac{M_{model}}{GPU_{memory} \times K_{avail}} \right\rceil = TP_{min} \tag{6}$$

Table 5: Variables for minimum GPU count calculation

Variable	Definition	Unit	Default	Notes
$GPU_{count_{min}}^{model=1}$	Minimum required GPUs per one model instance	GPUs	–	Minimum number of GPUs needed to place model weights. I.e., this also means minimum TP value (TP_{min}). TP (multiplier Z) can be consciously increased for throughput growth/latency reduction if topology allows (NVLink/NVSwitch)
M_{model}	Memory required for model weights	GiB	–	See Section 3.1
GPU_{memory}	Available memory of one GPU	GiB	–	Example: 40 or 80 GiB for A100
K_{avail}	Available memory coefficient	float	0.9	Recommended to reserve 10% of GPU memory for system needs and activations

4.2 Number of Model Instances per Server

$$N^{model}_{TP=TP_{min}} = \left\lfloor \frac{GPU^{server}_{count}}{GPU^{model=1}_{count_{min}}} \right\rfloor \tag{7}$$

$$GPU^{model=1}_{count_{TP=Z}} = Z \times GPU^{model=1}_{count_{min}} \tag{8}$$

$$N^{model}_{TP=Z} = \left\lfloor \frac{GPU^{server}_{count}}{GPU^{model=1}_{count_{TP=Z}}} \right\rfloor = \left\lfloor \frac{N^{model}_{TP=TP_{min}}}{Z} \right\rfloor \tag{9}$$

Table 6: Variables for model instances per server calculation

Variable	Definition	Unit	Default	Notes
$N^{model}_{TP=TP_{min}}$	Number of model instances per server at minimum required TP	int	–	Rounded down
$N^{model}_{TP=Z}$	Number of model instances per server accounting for TP multiplier (Z)	int	–	Rounded down
Z	TP multiplier	int	1	Additional multiplier for TP, increasing number of used GPUs for throughput increase. $Z = 1, 2, 4, \ldots$
GPU^{server}_{count}	Number of GPUs on one server	GPUs	–	Example: 8 GPU A100
$GPU^{model=1}_{count_{min}}$	Number of GPUs per one model instance	GPUs	–	See Section 4.1
$GPU^{model=1}_{count_{TP=Z}}$	Number of GPUs per one model instance accounting for TP multiplier (Z)	GPUs	–	–

4.3 Free GPU Memory for KV-Cache per Model Instance

Note: KV-cache must necessarily be stored on the same GPUs where the model is deployed.

$$GPU^{model}_{memoryForKV} = GPU^{model=1}_{count_{TP=Z}} \times GPU_{memory} \times K_{avail} - M_{model} \tag{10}$$

Table 7: Variables for free KV-cache memory calculation

Variable	Definition	Unit	Default	Notes
$GPU^{model}_{memoryForKV}$	Free memory for KV-cache	GiB	–	Remaining space for KV-cache
$GPU^{model=1}_{count_{TP=Z}}$	Number of GPUs per one model instance accounting for TP multiplier (Z)	GPUs	–	See Section 4.1
GPU_{memory}	Available memory of one GPU	GiB	–	Example: 40 or 80 GiB for A100
K_{avail}	Available memory coefficient	float	0.9	Recommended to reserve 10% of GPU memory for system needs and activations
M_{model}	Memory required for model weights	GiB	–	See Section 3.1

4.4 Maximum Theoretical Number of Parallel Sessions

$$S_{TP=Z} = \left\lfloor \frac{GPU^{model}_{memoryForKV}}{M^{S=1}_{KV}} \right\rfloor \tag{11}$$

Table 8: Variables for maximum parallel sessions calculation

Variable	Definition	Unit	Default	Notes
$S_{TP=Z}$	Maximum theoretical number of parallel sessions accounting for TP multiplier (Z)	sessions	–	–
$GPU^{model}_{memoryForKV}$	Free memory for KV-cache	GiB	–	See Section 4.3
$M^{S=1}_{KV}$	KV-cache per 1 session	GiB	–	See Section 3.2

5 Determining Number of Servers by GPU Memory

5.1 Throughput of One Server by Memory

$$S_{server} = N_{TP=Z}^{model} \times S_{TP=Z} \tag{12}$$

Table 9: Variables for server memory throughput calculation

Variable	Definition	Unit	Default	Notes
S_{server}	Number of sessions simultaneously supported by server	sessions	–	Number of parallel sessions per server
$N_{TP=Z}^{model}$	Number of model instances per server accounting for Tensor Parallelism	int	–	See Section 4.2
$S_{TP=Z}$	Number of parallel sessions accounting for Tensor Parallelism = Z	sessions	–	See Section 4.4

5.2 Number of Servers by Memory

$$Servers_{count}^{mem} = \left\lceil \frac{S_{sim}}{S_{server}} \right\rceil \tag{13}$$

Table 10: Variables for server count by memory calculation

Variable	Definition	Unit	Default	Notes
$Servers_{count}^{mem}$	Number of servers by memory	servers	–	–
S_{sim}	Peak number of simultaneous employee sessions	sessions	–	See Section 2.1
S_{server}	Server throughput	sessions	–	See Section 5.1

6 Determining Number of Servers by GPU Computational Throughput

6.1 Throughput of One Model Instance

Approximate estimate of computational cost per token for forward pass (inference): $2 \times N$, for forward + backward pass (training): $6 \times N$.

$$FPS = 2 \times P \times 10^9 \tag{14}$$

Table 11: Variables for FLOPs per token calculation

Variable	Definition	Unit	Default	Notes
FPS	Number of FLOPs per 1 token	FLOPs/token	–	During forward pass, each model parameter participates in one multiplication and one addition per token as a result of matrix-vector multiplication (matrix=model weights, vector=token hidden state). This is 2 operations (FLOP). Summing over all parameters, we get $2 \times P$ FLOPs per token, where P is number of parameters. In formulas below, attention contribution is added to basic weight-based FPS estimate, which becomes more significant with increasing context length
P	Number of model parameters	int	–	Example: 7B, 13B, 70B. For MoE, number of active parameters is used

$$K_{batch} = \frac{S_{TP=Z}}{S_{TP=TP_{min}}} \times \frac{S_{TP=TP_{min}} + C}{S_{TP=Z} + C} \tag{15}$$

Table 12: Variables for batch coefficient calculation

Variable	Definition	Unit	Default	Notes
K_{batch}	Throughput increase coefficient due to Tensor Parallelism	float	–	Throughput gain coefficient (see Appendix A) – upper estimate
GPU_{count}^{model}	Number of GPUs per one model instance	GPUs	–	See Section 4.1
C	Batch "saturation" coefficient	float	–	$C = \frac{t_{fix}}{t_{lin}}$. Larger C means stronger influence of fixed overhead. Estimate: 4–16
$S_{TP=Z \times GPU_{count}^{model}}$	Number of parallel sessions accounting for Tensor Parallelism	sessions	–	–

Indicators Th_{pf} and Th_{dec} depend on many factors, e.g., model, GPU type, applied parallelism methods, quantization, and used frameworks. General analytical calculation formula (without experiments, https://lmsys.org/blog/2025-09-21-petit-amdgpu/, https://habr.com/ru/companies/yandex/articles/801119/):

$$T_{dec} = A + MRT \tag{16}$$

$$Th_{pf}^{analyt} \approx \frac{F_{count}^{model} \times \eta_{pf} \times K_{batch}}{FPS + 4 \times L \times H \times SL} \tag{17}$$

$$Th_{dec}^{analyt} \approx \frac{F_{count}^{model} \times \eta_{dec} \times K_{batch}}{FPS + 4 \times L \times H \times (SL + (T_{dec} - 1)/2)} \tag{18}$$

Note: For approximate estimation without access to target hardware, it is recommended to first select Th_{pf} and Th_{dec} from public benchmarks (vendor/model/inference engine documentation, MLPerf, articles with reproducible settings), and use analytical FLOPS formulas as a last resort. For correctness, two inference phases should be distinguished: prefill (input context processing) and decode (step-by-step generation). If experiments or real tasks exist, priority should be given to using empirical values obtained from them.

Ideally, fix separately Th_{pf}^{empir} (input/prefill) and Th_{dec}^{empir} (output/decode) for comparable context lengths and batch sizes.

Table 13: Variables for throughput calculation

Variable	Definition	Unit	Default	Notes
Th_{pf}^{empir}	Empirical throughput indicator for prefill phase for 1 model instance	tokens/sec	–	Take from public benchmarks as input/prefill (preferably for comparable SL and batch size)
Th_{dec}^{empir}	Empirical throughput indicator for decode phase for 1 model instance	tokens/sec	–	Take from public benchmarks as output/decode
Th_{pf}^{analyt}	Analytical throughput indicator for prefill phase for 1 model instance	tokens/sec	–	Analytical estimate – ideally should not be used
Th_{dec}^{analyt}	Analytical throughput indicator for decode phase for 1 model instance	tokens/sec	–	Analytical estimate – ideally should not be used
F_{count}^{model}	Peak GPU performance accounting for their number per 1 model instance	FLOPs/sec	–	Per specification, e.g., 312 TFLOPS for A100
η_{pf}	Efficiency coefficient of peak power usage on prefill phase	float	0.20	Estimate: 0.15–0.30. Conservative: 0.15–0.20
η_{dec}	Efficiency coefficient of peak power usage on decode phase	float	0.15	Estimate: 0.10–0.25. Conservative: 0.10–0.15
K_{batch}	Throughput increase coefficient due to Tensor Parallelism	float	–	See Section 4.4
FPS	Number of FLOPs per 1 token	FLOPs/token	–	By formula above
L	Number of model layers	layers	–	See Section 3.2
H	Hidden state dimension	int	–	See Section 3.2
SL	Context window sequence length	tokens	–	See Section 3.2
T_{dec}	Number of tokens generated by model in decode phase per request	tokens	–	Calculated for generation process

6.2 Average Number of Simultaneous Requests per Second per Model Instance

$$C^{model} = \frac{1}{\frac{TS^{S=1}_{prp=5}}{Th_{pf}} + \frac{T_{dec}}{Th_{dec}}} \tag{19}$$

Table 14: Variables for requests per second calculation

Variable	Definition	Unit	Default	Notes
C^{model}	Requests per second per one model instance	requests/sec	–	Required for determining server throughput
$TS^{S=1}_{prp=5}$	Number of tokens processed in prefill phase per request	tokens	–	See Section 3.2
Th_{pf}	Throughput of prefill phase for 1 model instance	tokens/sec	–	Choose as Th^{empir}_{pf} or Th^{analyt}_{pf} (see Section 6.1)
T_{dec}	Number of tokens generated by model in decode phase per request	tokens	–	See Section 6.1
Th_{dec}	Throughput of decode phase for 1 model instance	tokens/sec	–	Choose as Th^{empir}_{dec} or Th^{analyt}_{dec} (see Section 6.1)

6.3 Final Throughput of One Server

$$Th^{comp}_{server} = N^{model}_{TP=Z} \times C^{model} \tag{20}$$

Table 15: Variables for final server throughput calculation

Variable	Definition	Unit	Default	Notes
Th^{comp}_{server}	Server throughput	requests/sec	–	–
$N^{model}_{TP=Z}$	Number of model instances per server accounting for Tensor Parallelism	int	–	See Section 4.2
C^{model}	Requests per second per one model instance	requests/sec	–	See Section 6.2

6.4 Number of Servers by Throughput

$$Servers_{count}^{comp} = \left\lceil \frac{S_{sim} \times R \times K_{SLA}}{Th_{server}^{comp}} \right\rceil \tag{21}$$

Table 16: Variables for server count by throughput calculation

Variable	Definition	Unit	Default	Notes
$Servers_{count}^{comp}$	Number of servers by computational power	servers	–	–
S_{sim}	Peak number of simultaneous employee sessions	sessions	–	See Section 2.1
R	Requests per second per session	requests/sec	0.02	Chat: 1/30–60 sec → 0.017–0.033. For batch – higher
K_{SLA}	Safety margin coefficient for SLA	float	1.25	Margin for SLA and uncertainty: 1.25–2.0
Th_{server}^{comp}	Server throughput	requests/sec	–	See Section 6.3

7 Verification of Selected Server Configurations by TTFT and e2eLatency

7.1 Time To First Token (TTFT) Calculation

$$TTFT_{analyt} = \frac{SL}{Th_{pf}} + \frac{1}{Th_{dec}} \tag{22}$$

Table 17: Variables for TTFT calculation

Variable	Definition	Unit	Default	Notes
$TTFT_{analyt}$	Time to first token generation at decode stage	sec	–	–
SL	Context window sequence length	tokens	–	See Section 3.2
Th_{pf}	Throughput of prefill phase for one model instance	tokens/sec	–	See Section 6.1
Th_{dec}	Throughput of decode phase for one model instance	tokens/sec	–	See Section 6.1

7.2 Latency Indicator Determination

$$GenerationTime_{analyt} = \frac{T_{dec}}{Th_{dec}} \tag{23}$$

Table 18: Variables for generation time calculation

Variable	Definition	Unit	Default	Notes
$GenerationTime_{analyt}$	Time to generate all tokens at decode stage	sec	–	–
T_{dec}	Number of tokens generated by model in decode phase per request	tokens	–	See Section 6.1
Th_{dec}	Model throughput at decode stage	tokens/sec	–	See Section 6.1

$$e2eLatency_{analyt} = TTFT_{analyt} + GenerationTime_{analyt} \tag{24}$$

Table 19: Variables for end-to-end latency calculation

Variable	Definition	Unit	Default	Notes
$e2eLatency_{analyt}$	End-to-End Latency indicator without network delays	sec	–	–
$TTFT_{analyt}$	Time to first token generation at decode stage	sec	–	See Section 7.1
$GenerationTime_{analyt}$	Time to generate all tokens at decode stage	sec	–	See Section 7.2

7.3 Comparison of TTFT and e2eLatency with Target SLA Values

$$TTFT_{SLA} \geq TTFT_{analyt} \tag{25}$$

$$e2eLatency_{SLA} \geq e2eLatency_{analyt} \tag{26}$$

Table 20: Variables for SLA comparison

Variable	Definition	Unit	Default	Notes
$e2eLatency_{SLA}$	End-to-End Latency indicator without network delays	sec	–	Determined based on SLA requirements
$TTFT_{SLA}$	Time to first token generation at decode stage	sec	–	Determined based on SLA requirements

Note: If conditions are not met, see Appendix B.

8 Determining Final Number of Servers

$$Servers_{final} = \max\left(Servers_{count}^{mem}, Servers_{count}^{comp}\right) \tag{27}$$

Table 21: Variables for final server count calculation

Variable	Definition	Unit	Default	Notes
$Servers_{final}$	Final number of servers	servers	–	–
$Servers_{count}^{mem}$	Number of servers by memory	servers	–	See Section 5.2
$Servers_{count}^{comp}$	Number of servers by computational power	servers	–	See Section 6.4

References

[Vaswani et al.(2017)] Vaswani, A., Shazeer, N., Parmar, N., et al. (2017). Attention Is All You Need. *Advances in Neural Information Processing Systems (NeurIPS)*. https://arxiv.org/abs/1706.03762

[Kaplan et al.(2020)] Kaplan, J., McCandlish, S., Henighan, T., et al. (2020). Scaling Laws for Neural Language Models. *arXiv preprint arXiv:2001.08361*. https://arxiv.org/abs/2001.08361

[Hoffmann et al.(2022)] Hoffmann, J., Borgeaud, S., Mensch, A., et al. (2022). Training Compute-Optimal Large Language Models. *Advances in Neural Information Processing Systems (NeurIPS)*. https://arxiv.org/abs/2203.15556

[Shoeybi et al.(2019)] Shoeybi, M., Patwary, M., Puri, R., et al. (2019). Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. *arXiv preprint arXiv:1909.08053*. https://arxiv.org/abs/1909.08053

[Kwon et al.(2023)] Kwon, W., Li, Z., Zhuang, S., et al. (2023). Efficient Memory Management for Large Language Model Serving with PagedAttention. *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP)*. https://arxiv.org/abs/2309.06180

[Frantar et al.(2023)] Frantar, E., Ashkboos, S., Hoefler, T., & Alistarh, D. (2023). GPTQ: Accurate Post-Training Quantization for Generative Pre-trained Transformers. *International Conference on Learning Representations (ICLR)*. https://arxiv.org/abs/2210.17323

[Shazeer et al.(2017)] Shazeer, N., Mirhoseini, A., Maziarz, K., et al. (2017). Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer. *International Conference on Learning Representations (ICLR)*. https://arxiv.org/abs/1701.06538

[Aminabadi et al.(2022)] Aminabadi, R.Y., Rajbhandari, S., Zhang, M., et al. (2022). DeepSpeed Inference: Enabling Efficient Inference of Transformer Models at Unprecedented Scale. *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. https://arxiv.org/abs/2207.00032

[Narayanan et al.(2021)] Narayanan, D., Shoeybi, M., Casper, J., et al. (2021). Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM. *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. https://arxiv.org/abs/2104.04473

[NVIDIA(2021)] NVIDIA. (2021). Scaling Language Model Training to a Trillion Parameters Using Megatron. *NVIDIA Developer Blog*. https://developer.nvidia.com/blog/scaling-language-model-training-to-a-trillion-parameters-using-megatron/

[MLCommons(2024)] MLCommons. (2024). MLPerf Inference Benchmark Suite – Datacenter. https://mlcommons.org/benchmarks/inference-datacenter/

[Databricks(2023)] Databricks. (2023). LLM Inference Performance Engineering: Best Practices. *Databricks Blog*. https://www.databricks.com/blog/llm-inference-performance-engineering-best-practices

[Gorb(2024)] Gorb, R. (2024). Accelerating LLM Inference. *Habr (Yandex)*. https://habr.com/ru/companies/yandex/articles/801119/

[LMSYS(2025)] LMSYS Org. (2025). Serving LLMs on AMD GPUs with SGLang. *LMSYS Blog*. https://lmsys.org/blog/2025-09-21-petit-amdgpu/

[Wharton(2025)] Wharton Generative AI Labs. (2025, June). The Decreasing Value of Chain of Thought in Prompting. https://gail.wharton.upenn.edu/research-and-insights/tech-report-chain-of-thought/

[CMU(2025)] CMU/arXiv. (2025, May). Incorporating Token Usage into Prompting Strategy Evaluation (Big-O_tok). https://arxiv.org/html/2505.14880v1

[Madaan et al.(2023)] Madaan, A., et al. (2023). Self-Refine: Iterative Refinement with Self-Feedback. *NeurIPS 2023*. https://arxiv.org/abs/2303.17651

[LangChain(2025)] LangChain. (2025, February). Benchmarking Single Agent Performance. https://blog.langchain.com/react-agent-benchmarking/

[AIMultiple(2026)] AIMultiple. (2026, March). Multi-Agent Frameworks Benchmark. https://aimultiple.com/multi-agent-frameworks

[arXiv(2025)] arXiv. (2025, November). TeaRAG: A Token-Efficient Agentic RAG Framework. https://arxiv.org/html/2511.05385v1

[OpenAI(2025)] OpenAI. (2025). Function Calling Guide. https://developers.openai.com/api/docs/guides/function-calling/

[CodeAnt(2026)] CodeAnt. (2026). How Poor Tool Calling Behavior Increases LLM Cost and Latency. https://www.codeant.ai/blogs/poor-tool-calling-llm-cost-latency

A Derivation of Throughput Gain Calculation from Tensor Parallelism

Parameters:

- G – memory of one GPU (in GB)
- $M_{w,total}$ – memory for model weights entirely, if it were on one card
- $M_{act,total}$ – memory for activations per request (batch=1) without TP, i.e., memory for KV-cache
- K_{avail} – fraction of memory we leave available (e.g., 0.9 = 90%)
- T – number of GPUs used for one model with Tensor Parallelism

Formula derivation:

1. Weights on one GPU:
$$M_{w,per_gpu}(T) \approx \frac{M_{w,total}}{T} \tag{28}$$

2. Activations on one GPU for one request:
$$M_{act,per_gpu}(T) \approx \frac{M_{act,total}}{T} \tag{29}$$

3. Available memory for batch on one GPU:
$$M_{avail,per_gpu}(T) \approx \left(G - \frac{M_{w,total}}{T}\right) \times K_{avail} \tag{30}$$

4. Maximum batch per one "model" (group of T GPUs):
   In exact (integer) form:
$$BS_{max}(T) \approx \left\lfloor \frac{M_{avail,per_gpu}(T)}{M_{act,per_gpu}(T)} \right\rfloor \tag{31}$$
   Substituting expressions above:
$$BS_{max}(T) \approx \left\lfloor \frac{\left(G - \frac{M_{w,total}}{T}\right) \times K_{avail}}{\frac{M_{act,total}}{T}} \right\rfloor \tag{32}$$
   If we ignore floor($\cdot$) for analytical form, we get:
$$BS_{max}(T) \approx \frac{\left(G - \frac{M_{w,total}}{T}\right) \times K_{avail} \times T}{M_{act,total}} \tag{33}$$

5. Batch time model

   We use a simple linear model:

   - Batch processing time of size BS:

   $$t(BS) = t_{fix} + t_{lin} \times BS \tag{34}$$

   where:
   - t_{fix} – fixed time for launch, communications, etc.
   - t_{lin} – variable time per request in batch (linear in B)

   - Throughput (number of requests per unit time):

   $$Th(BS) = \frac{BS}{t(BS)} = \frac{BS}{t_{fix} + t_{lin} \times BS} \tag{35}$$

   Introduce dimensionless parameter:

   $$C = \frac{t_{fix}}{t_{lin}} \tag{36}$$

   Then, factoring out, we can write:

   $$Th(BS) \propto \frac{BS}{C + BS} \tag{37}$$

   $\frac{1}{t_{lin}}$ can be omitted, as we are only interested in ratios through $Th(BS_1)/Th(BS_2)$

6. Throughput gain coefficient relative to TP = 1

   Let:

   - at TP = 1, maximum batch: $BS_1 = BS_{max}(1)$
   - at TP = T, maximum batch: $BS_T = BS_{max}(T)$

   Then corresponding throughput (up to common constant) equals:

   $$Th_1 \propto \frac{BS_1}{C + BS_1}, \quad Th_T \propto \frac{BS_T}{C + BS_T} \tag{38}$$

7. Throughput gain coefficient relative to TP = 1

   $$K_{Th}(T) = \frac{Th_T}{Th_1} \tag{39}$$

   Substituting:

   $$K_{Th}(T) = \frac{\frac{BS_T}{C+BS_T}}{\frac{BS_1}{C+BS_1}} \tag{40}$$

   Simplifying:

   $$K_{Th}(T) = \frac{BS_T}{BS_1} \times \frac{BS_1 + C}{BS_T + C} \tag{41}$$

   This is the pure throughput growth coefficient relative to $T = 1$, which:

   - accounts for how T changes maximum batch through memory constraints
   - accounts for the fact that gain from batch growth cannot increase infinitely due to fixed overhead ($C = \frac{t_{fix}}{t_{lin}}$)

B Recommendations for Aligning Configuration with Target TTFT and e2eLatency Values

If calculated TTFT and/or e2eLatency values exceed target thresholds ($TTFT_{target}$, $e2eLatency_{target}$), it is recommended to sequentially apply the following measures – from least invasive to most radical.

Note: These recommendations are not exhaustive and may be supplemented or used as needed.

B.1 Increase TP Multiplier (Z)

What it provides: Growth of throughput of one model instance (Th_{pf}, Th_{dec}) by distributing computations across more GPUs.

Effect on TTFT: Decreases proportionally to growth of Th_{pf}.

Effect on e2eLatency: Decreases proportionally to growth of Th_{dec}.

Limitations:

- Z is limited by number of GPUs per server (Z_{max})
- Each doubling of Z halves the number of model instances per server – may require more servers to cover memory and throughput load
- Scaling efficiency $\eta_{TP} < 1$ (communication overhead between GPUs), especially when using PCIe instead of NVLink

When to apply: First priority. This is the standard mechanism of the methodology.

B.2 Apply More Aggressive Quantization

What it provides: Reduction of bytes per parameter (B_{quant}) and per KV-cache state value (B_{state}). Reduces data volume read from GPU memory per token, which increases Th_{pf} and Th_{dec}.

Examples:

- FP16 → FP8: throughput gain ∼1.5–2×
- FP16 → INT4 (GPTQ/AWQ): throughput gain ∼2–3×

Effect on TTFT: Decreases due to growth of Th_{pf}.

Effect on e2eLatency: Decreases due to growth of Th_{dec}.

Limitations:

- Possible degradation of generation quality (especially with INT4 for complex reasoning tasks)
- Not all models and frameworks support all quantization levels
- Quality validation required on target use-cases

When to apply: If increasing Z is impossible or insufficient, and quality impact verification is acceptable.

B.3 Reduce Context Length (SL)

What it provides: Reduction of data volume processed in prefill phase, which directly reduces TTFT. Also reduces attention contribution to computational cost per token.

Ways to reduce:

- Reduce system prompt (SP): remove redundant instructions, use compressed formulations
- Reduce dialogue depth (N_{prp}): limit number of stored messages in context
- Apply dialogue history summarization instead of passing full context
- Use RAG with limited number of retrieved fragments

Effect on TTFT: Direct reduction ($TTFT \propto SL/Th_{pf}$).

Effect on e2eLatency: Indirect reduction through TTFT decrease.

Limitations:

- May reduce model response quality (less context – less relevant information)
- Requires changes in application logic

When to apply: If TTFT is the main bottleneck (e2eLatency may be acceptable).

B.4 Reduce Generation Volume (T_{out})

What it provides: Reduction of decode phase time, which is usually the dominant component of e2eLatency.

Ways to reduce:

- Limit max_tokens in request to model
- Reduce MRT (reasoning budget): switch model to mode without reasoning or set stricter limit
- Configure prompt so model generates more compact responses

Effect on TTFT: No effect.

Effect on e2eLatency: Direct reduction ($GenerationTime \propto T_{out}/Th_{dec}$).

When to apply: If e2eLatency is the main bottleneck, while TTFT is acceptable.

B.5 Change Model

What it provides: Model with fewer parameters (P) has proportionally higher throughput with same computational resources. MoE models have fewer active parameters ($P_{active} < P_{total}$), which gives similar effect.

Transition examples:

- Dense 70B → Dense 32B: ∼2× throughput growth
- Dense 70B → MoE 30B-A3B: ∼9× throughput growth (by active parameters)

- Dense 70B → Dense 7B: ∼10× throughput growth

Effect on TTFT: Decreases proportionally to P_{active} reduction.

Effect on e2eLatency: Decreases proportionally to P_{active} reduction.

Limitations:

- Smaller model may not provide required quality on target tasks
- Full quality revalidation required

When to apply: If previous measures are exhausted or insufficient.

B.6 Use More Performant Hardware

What it provides: GPUs with higher memory bandwidth and/or more TFLOPS provide higher throughput, all else equal.

Examples:

- H100 → H200: ∼40–45% throughput growth (due to memory bandwidth 4.8 vs 3.35 TB/s)
- H100 → B200: ∼2.5× throughput growth

Effect on TTFT: Decreases due to growth of Th_{pf}.

Effect on e2eLatency: Decreases due to growth of Th_{dec}.

Limitations:

- Increased CapEx
- Equipment availability and delivery timelines
- Requires recalculation of entire methodology with new GPU parameters

When to apply: During procurement planning or when SLA cannot be met on existing hardware.

B.7 Reconsider Target SLA Values

What it provides: Alignment of requirements with real infrastructure capabilities.

Recommendations:

- Agree with business stakeholder which TTFT and e2eLatency values are truly critical for UX
- Differentiate SLA by segments (e.g., stricter for VIP clients, softer for internal users)
- Consider that in streaming mode, user starts seeing text from TTFT moment, so high e2eLatency is less critical with low TTFT

When to apply: If technical cost of meeting SLA is disproportionate to business value.

B.8 Measure Selection Matrix

Table 22: Recommendation selection matrix by situation

Situation	TTFT	e2eLatency	Priority Measures
Both indicators exceed targets	✗	✗	$1 \to 2 \to 5 \to 6$
Only TTFT exceeds	✗	✓	$3 \to 1 \to 2$
Only e2eLatency exceeds	✓	✗	$4 \to 1 \to 2$
Both acceptable, but no margin	✓(tight)	✓(tight)	$2 \to 3 \to 4$ (preventive)

Note: Recommendations 1–2 change parameters within current methodology and require full recalculation of Sections 3–7.

Recommendations 3–4 change input load parameters (Section 2) and also require full recalculation.

Recommendations 5–6 change input model and equipment parameters, which requires methodology recalculation from scratch.

Recommendation 7 changes target SLA values and requires only re-verification per this section.

C Recommendations for Token Load Estimation Strategy Depending on AI Solution Architecture

C.1 Connection with Main Methodology Formulas

The main methodology defines session context length by formula (Section 3.2):

$$TS = SP + N_{prp} \times (Prp + MRT + A) \tag{42}$$

where SP – system prompt (default 1,000 tokens), N_{prp} – number of request-response pairs in session (default 5), Prp – user request (200), MRT – reasoning budget (4,096), A – model response (400).

Formula (3.2) correctly describes simple dialogue scenario: one user request → one LLM call → one response. However, modern AI systems increasingly use agent patterns (ReAct, multi-agent teams), reasoning strategies (Chain-of-Thought, Self-Refine), and composite pipelines (RAG, function calling), where one user request may generate from 2 to 20+ internal LLM calls. Formula (3.2) does not explicitly account for this.

This appendix shows how to adapt formula (3.2) parameters for various architectural patterns, and provides a pattern catalog with empirical token load data.

C.2 What Changes in Agent Architectures

Formula (3.2) operates on five parameters. Table 23 shows which remain unchanged and which require correction.

Table 23: Parameter comparison: simple chatbot vs. agent architecture

Parameter	Simple chatbot	Agent architecture	What changes
SP	Model instruction: ∼1,000 tokens	Instruction + tool descriptions + few-shot examples	Grows by 300–2,000 due to tool definitions
Prp	User request: ∼200 tokens	Request + RAG context + tool call results	Grows by 500–5,000 due to insertions
MRT	Reasoning budget: 0–4,096	Same budget spent on each step	No change in per-call calculation
A	Model response: ∼400 tokens	May include tool_call JSON, intermediate reasoning	Grows slightly (50–200)
N_{prp}	5 request-response pairs in session	Same number of user requests	No change
K_{calls} (new)	1 (one LLM call per request)	3–20 internal LLM calls per user request	Key multiplier, absent in (3.2)

C.3 Generalized Formula for Agent Architectures

To use formula (3.2) for agent scenarios, replace original parameters with their effective values:

$$SP_{eff} = SP + SP_{tools} + C_{rag_static} \tag{43}$$

where SP_{tools} – total volume of tool descriptions (tool definitions) embedded in system prompt; C_{rag_static} – static RAG context, if loaded once at session initialization.

$$Prp_{eff} = Prp + C_{rag_dynamic} \tag{44}$$

where $C_{rag_dynamic}$ – dynamic RAG context, loaded per specific user request (typical values: 500–5,000 tokens).

$$A_{eff} = A + A_{tool} \tag{45}$$

where A_{tool} – additional tokens in model response for tool calls (tool_call JSON, usually 50–200 tokens).

The main difference of agent architectures – internal call multiplier K_{calls}. One user request generates K_{calls} sequential LLM calls. Each step consumes full context (SP + history + new input). Therefore, formula (3.2) takes form:

$$TS_{agent} = SP_{eff} + N_{prp} \times K_{calls} \times (Prp_{eff} + MRT + A_{eff}) \tag{46}$$

When $K_{calls} = 1$ and zero agent overheads, formula (46) reduces to formula (3.2), ensuring backward compatibility.

Important: Formula (46) gives upper estimate of context length. For peak GPU memory calculation, use (46) and substitute result as SL in KV-cache formula (Section 3.2).

C.4 Architectural Pattern Catalog

Below are main patterns and their characteristics from token load perspective. For each pattern, typical K_{calls} values and overheads are indicated, which should be substituted into formula (46).

C.4.1 Single-turn Chatbot (Basic Case)

One request – one response. $K_{calls} = 1$. Overheads minimal: only system prompt (200–500 tokens). This is basic scenario, to which formula (3.2) applies directly without modifications.

C.4.2 RAG (Retrieval-Augmented Generation)

RAG adds relevant fragment extraction step from knowledge base before generation.

In simplest implementation $K_{calls} = 1$ (search itself performed by embedding model, not LLM), but Prp_{eff} significantly increases due to insertion of retrieved fragments into context.

Typically 1–5 fragments of 500–1,000 tokens added, i.e., C_{rag} = 1–5K tokens.

In agentic RAG, LLM itself decides when and what to search, performing multiple search iterations. TeaRAG study (November 2025) showed that agentic RAG without optimization generates 2–3× more output tokens; K_{calls} in such case = 2–4.

C.4.3 Chain-of-Thought (CoT) and Reasoning Models

CoT does not increase K_{calls} ($K_{calls} = 1$), but significantly increases output tokens.

Wharton AI Labs study (June 2025) recorded that CoT requests require 35–600% more tokens compared to direct requests.

For CoT-SC (Self-Consistency), where p parallel reasoning chains generated, load grows linearly: $K_{calls} = p$. Big-O_tok study (CMU, May 2025) showed that CoT-SC with $p = 5$ consumes 15–40× more tokens than basic request.

Reasoning models (o1, o3, DeepSeek-R1) generate hidden "thinking tokens" of 8–25K tokens per request. These tokens are not returned to user, but consume GPU resources during decoding and should be accounted via MRT parameter.

C.4.4 Self-Refine / Iterative Improvement

Self-Refine pattern (Generation → Evaluation → Refine) performs 2–3 iterations, each containing 2 LLM calls (evaluation + rewriting). Madaan et al. (2023) showed that 3 iterations improve quality by ∼20%, but consume 5–7× more tokens. $K_{calls} = 1 + 2 \times n_{iter} \approx 5$–7.

C.4.5 ReAct (Agent with Tools)

ReAct agent executes Thought → Action → Observation cycle until goal achieved. Each cycle – separate LLM call with accumulated context. Moreover, context grows quadratically: each next call includes all previous history.

LangChain benchmark (February 2025) showed that tasks with long trajectories (3–6 steps) degrade with context growth to 47–77K tokens. Halder experiments (2026) showed ∼4.5× token growth when task complexity doubled.

Additional overhead created by tool descriptions (tool definitions): per OpenAI data, each tool description occupies 300–500 tokens and transmitted in each LLM call. With 10 tools, this is ∼5,000 tokens overhead per call (SP_{tools}). Typically $K_{calls} = 3$–10.

C.4.6 Multi-Agent Systems

Multi-agent frameworks (CrewAI, AutoGen, LangGraph) orchestrate several specialized agents. Each agent performs own LLM calls, and coordination between them adds additional load.

AIMultiple benchmark (2026) on 750 runs revealed key problem: sequential pipeline leads to exponential token growth, as output of each agent becomes input of next. CrewAI with 5 agents consumed 1.35M tokens per task, whereas LangGraph with dynamic routing – 13.6K tokens (∼100× difference).

Typically $K_{calls} = N_{agents} \times K_{steps_per_agent}$. With 3–5 agents, each performing 2–4 calls, $K_{calls} = 6$–20.

C.4.7 Function Calling / Tool Use

Tool calling – multi-step process: (1) LLM generates function call, (2) system executes it, (3) result passed back to LLM. Each tool call – minimum 2 LLM calls. Per OpenAI data, function descriptions embedded in system message and billed as input tokens.

C.5 Context Compounding Effect

In agent systems, LLM calls are not independent: each subsequent call includes all previous history. This leads to quadratic growth of input tokens. If each step generates s response tokens, then total input token consumption over n steps:

$$T_{in_total} \approx n \times C_{sys} + \frac{n \times (n-1)}{2} \times s \tag{47}$$

This explains why AIMultiple benchmark showed 100× difference between sequential (CrewAI) and dynamic (LangGraph) orchestrators: in first case, full context transmitted to each next agent, in second – only relevant fragment transmitted.

Connection with formula (46): Formula (46) gives linear upper estimate. For systems with long chains ($K_{calls} > 5$), it is recommended to additionally use formula (47) for refining peak consumption, especially when planning KV-cache.

C.6 Summary Pattern Table

Table 24 contains recommended ranges for substitution into formula (46) and overall token load multiplier relative to basic scenario.

Table 24: Summary characteristics of architectural patterns. Token multiplier shows aggregate increase relative to basic single-turn scenario.

Pattern	K_{calls}	SP_{tools}	C_{rag}	A_{tool}	Token Mult.	Note
Single-turn chat	1	0	0	0	1×	= formula (3.2)
RAG	1–2	0–200	1–5K	0	1.5–3×	Search + generation
Agentic RAG	2–4	0–200	1–5K ×N	0	3–8×	Multi-step RAG
CoT	1	0	0	0	2–7×	Growth via S_{out}
CoT-SC ($p = 5$)	5	0	0	0	15–40×	Majority voting
Reasoning models	1	0	0	0	3–10×	+8–25K thinking tokens
Self-Refine	5–7	0	0	0	5–7×	Generate → evaluate → rewri
Function calling	2–3	300–500	0	100–200	2–5×	plan → call → summarize
ReAct agent	3–10	500–2K	0–3K	100–200	5–15×	+ compounding (47)
Multi-agents	6–20+	500–2K	0–5K	100–200	10–100×	CrewAI, AutoGen, Swarm
Coding agent	5–15	1–3K	2–10K	200–500	10–30×	Aider, Cursor, Copilot

C.7 Calculation Examples

C.7.1 Example 1: Simple RAG Chat

Parameters: $SP = 1,000$, $SP_{tools} = 0$, $C_{rag} = 2,000$ (3 chunks $\times$ $\sim$670 tokens), $K_{calls} = 1$, $Prp = 200$, $MRT = 0$, $A = 400$, $N_{prp} = 5$.

$$\begin{aligned} SP_{eff} &= 1,000, \\ Prp_{eff} &= 200 + 2,000 = 2,200, \\ A_{eff} &= 400 \\ TS &= 1,000 + 5 \times 1 \times (2,200 + 0 + 400) = 14,000 \text{ tokens} \end{aligned}$$

By basic formula (3.2) without RAG: $TS = 4,000$. Increase: $\times 3.5$.

C.7.2 Example 2: ReAct Agent with Tools

Parameters: $SP = 1,000$, $SP_{tools} = 1,500$ (5 tools $\times$ 300), $C_{rag} = 1,000$, $K_{calls} = 5$, $Prp = 200$, $MRT = 4,096$, $A = 400$, $A_{tool} = 150$, $N_{prp} = 3$.

$$\begin{aligned} SP_{eff} &= 2,500, \\ Prp_{eff} &= 1,200, \\ A_{eff} &= 550 \\ TS &= 2,500 + 3 \times 5 \times (1,200 + 4,096 + 550) = 90,190 \text{ tokens} \end{aligned}$$

By formula (3.2): $TS = 14,888$. Increase: $\times 6.1$. With $TS_{max} = 128K$, session fits, but KV-cache requires $\sim 6\times$ more memory.

C.7.3 Example 3: Multi-Agent Team

3 agents $\times$ 4 calls $\rightarrow$ $K_{calls} = 12$. $SP = 1,000$, $SP_{tools} = 2,000$, $C_{rag} = 3,000$, $Prp = 200$, $MRT = 4,096$, $A = 400$, $A_{tool} = 200$, $N_{prp} = 2$.

$$\begin{aligned} SP_{eff} &= 3,000, \\ Prp_{eff} &= 3,200, \\ A_{eff} &= 600 \\ TS &= 3,000 + 2 \times 12 \times (3,200 + 4,096 + 600) = 192,504 \text{ tokens} \end{aligned}$$

This exceeds $TS_{max} = 128K \rightarrow SL = 128,000$. By formula (3.2): $TS = 10,392$. Potential increase: $\times 18.5$. System "hits" context window, requiring optimization strategies (see Section C.9).

C.7.4 Example 2B: ReAct Agent with Compounding

Use formula (47) to refine Example 2. $C_{sys} = SP_{eff} = 2,500$ per call, $s = A_{eff} = 550$ tokens per step, $n = K_{calls} = 5$:

$$T_{in_total} = 5 \times 2,500 + \frac{5 \times 4}{2} \times 550 = 12,500 + 5,500 = 18,000 \text{ tokens per request}$$
$$T_{out_total} = 5 \times 550 = 2,750$$
$$\text{Total} \approx 20,750 \text{ tokens per user request}$$

For comparison: basic chatbot – ∼650 input + 400 output = 1,050. Increase: ×20.

C.8 Impact on Throughput Calculation

Throughput formula from Section 5 of methodology uses SL (context length) to determine KV-cache size. Increasing SL directly reduces number of parallel sessions:

1. KV-cache per session (Section 3.2): $M_{KV} = f(SL)$ – proportional to context length. With $SL = 90,000$ instead of 14,000, memory requirement grows ∼6.4×.
2. Number of sessions per GPU (Section 5): $S_{max} = GPU_{avail}/M_{KV}$ – inversely proportional to SL.
3. Number of servers (Section 6): proportional to target RPS / (S_{max}× inverse latency).

SL growth increases both latency and GPU requirement.

Thus, substituting TS_{agent} from (46) instead of TS from (3.2) into KV-cache formula is the only change for all subsequent calculations. Other formulas (Sections 4–8) require no modification.

C.9 Strategies for Reducing Token Load

With high K_{calls} values, context may quickly reach TS_{max}. Below are practical strategies.

Table 25: Token load reduction strategies

Strategy	**Description**	**Which parameter reduced**	**Effect**
History compression	Summarization of previous agent steps instead of passing full history	Prp_{eff} (history)	3–5× reduction
Sliding window	Passing only last N steps, not entire chain	Effective K_{calls}	Fixes upper context bound
Lazy tool definitions loading	Passing descriptions of only relevant tools	SP_{tools}	50–80% SP_{tools} reduction
Chunking + reranking RAG	Limiting number and size of RAG chunks, two-stage filtering	C_{rag}	Control C_{rag} within 1–2K
Prompt caching	Caching immutable prompt part (SP + tool definitions) at inference engine level (vLLM prefix caching)	Latency, cost	Does not reduce SL, but reduces prefill time and cost
Sub-agent decomposition	Splitting task into subtasks with separate contexts	Peak SL	Several short contexts; peak KV-cache lower

C.10 Evaluation Algorithm and Measurement Recommendations

Parameter selection algorithm:

1. Determine system architectural pattern per Table 24.
2. From table, determine K_{calls} range and overheads (SP_{tools}, C_{rag}, A_{tool}).
3. If empirical data (logs) available – use real average values instead of tabular.
4. Compute TS_{agent} by formula (46). For $K_{calls} > 5$, additionally check by (47).
5. Apply $SL = \min(TS_{agent}, TS_{max})$ and substitute into KV-cache formula.
6. Continue calculation per main methodology (Sections 4–8) without changes.

Measurement recommendations:

1. **Inference server level logging.** vLLM and TGI provide metrics `prompt_tokens` and `completion_tokens` per request. Collect statistics over 100–500 representative scenarios.
2. **Orchestrator level profiling.** Frameworks LangChain, CrewAI, AutoGen provide callbacks with token tracking. Use `result.token_usage` for aggregated statistics.
3. **Margin buffer.** Add coefficient 1.2–1.5× for variability: some requests will require more steps and retries.
4. **Use p95 instead of mean.** For SLA-critical systems, better to use 95th percentile for K_{calls} to account for tail scenarios.

Recommendation: For initial calculation, use upper bound of K_{calls} from range. This ensures power margin. After production launch, adjust value based on monitoring data.